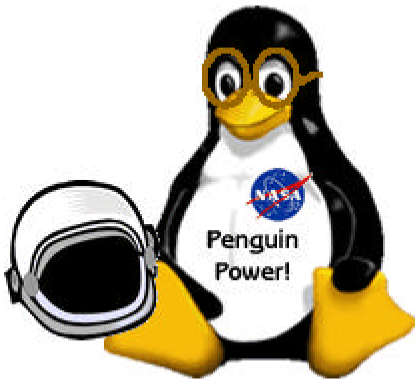
FlightLinux: A New Option for Spacecraft Embedded Computers
Paper A6P3

Patrick H. Stakem
Principal Investigator
QSS Group, Inc.
7404 Executive Place
Lanham, MD 20706

Abstract - The FlightLinux project has the stated goal of providing an on-orbit flight demonstration of the Linux operating system. This will result in a Technology Readiness Level of 7. The FlightLinux proof-of-concept demonstration is being done in conjunction with the on-orbit UoSat-12 mission, from Surrey Space Technology, Ltd. (SSTL) in the United Kingdom. The OMNI project of Code 588 at GSFC has procured a breadboard of the SSTL On-Board Computer (OBC), which is being used for testing. In addition, telecommunications facilities in Building 23 at GSFC allow direct communication with the UoSat-12 spacecraft.

This work was conducted under task NAS5-99124-297, with funding by the NASA Earth Science Technology Office (ESTO) Advanced Information Systems Technology (AIST) Program, NRA-99-OES-08. The work is conducted by personnel of QSS Group, Inc. in partnership with NASA/GSFC Codes 586 (Science Data Systems), 582 (Flight Software), and 588 (Advanced Architectures and Automation).

Because most of the effort in developing onboard computers for spacecraft involves adapting existing commercial designs, the logical next step is to adapt Commercial Off-The-Shelf (COTS) software, such as the Linux operating system. Given Linux, many avenues and opportunities become available. Web serving and file transfers become standard features. Onboard LAN and an onboard file system become "givens." Java is trivial to implement. Commonality with ground environments allows rapid migration of algorithms to the flight system, and tapping into the worldwide expertise of Linux developments provide a large pool of development and debugging talent. Full source for the operating system and drivers is available without cost.



Since we posted our goals of keeping the FlightLinux open source, within the meaning of the GNU license [1], we have had numerous offers of collaboration on the project. These include representatives of U.S. and European aerospace companies and individuals. The interest in the FlightLinux Project continues to grow due to the increasing exposure of our web site.

We are currently heavily into the development and testing of the FlightLinux port for the UoSat-12 onboard computer. We have received tremendous cooperation from SSTL and the Operating Missions as Nodes on the Internet (OMNI) Project, the owners of the UoSat-12 breadboard.

A significant challenge in the next phase of the project will be to complete the paper work, so that the FlightLinux software may be legally "exported" to the UoSat-12, a British platform. This process is expected to take 6 months and was unanticipated. The National Aeronautics and Space Administration (NASA) has confirmed that flight software is considered an export-controlled commodity on the Department of State's Munitions List. This issue is being worked with the Goddard Space Flight Center (GSFC) Legal Council. We still are hopeful we can release the FlightLinux software under the terms of the GPL (GNU Public License) as Open Software.

An initial build of the FlightLinux software has been running since March 2001.

The UoSat-12 breadboard arrived at the OMNI Lab and was tested successfully by the end of April 2001. The breadboard has an associated Windows-NT machine to load software via the Controller Area Network (CAN) bus or the asynchronous port and to provide debugging visibility. From QSS, we can access the breadboard facility via the "PC-Anywhere" software with the appropriate link security.

The initial FlightLinux software load is approximately 400,000 bytes in size. The nature of the UoSat-12 memory architecture at boot time limits the load size to less than 512,000 bytes. Four megabytes of memory is available after the loader, which is Read-Only Memory (ROM)-based, completes. The software load includes: 1) a routine to setup the environment and 2) a routine to decompress and start the Linux kernel. The kernel is the central portion of the operating system, a monolithic code entry. It controls process management, Input/Output, the file system, and other features. It provides an executive environment to the application programs, independent of the hardware.

Extensive customization of the SETUP routine, written in assembly language, was required. This routine in its original form relies on Basic Input/Output System (BIOS) calls to discover and configure hardware. In the UoSat configuration, there is no BIOS function, therefore, these sections were replaced with the equivalent code. Sections of SSTL code were added to configure the unique hardware of the UoSat computer. The SETUP routine then configures the processor for entry to Protected Mode and invokes the decompression routine for the kernel. The SETUP routine is approximately 750 bytes in length and represents the custom portion of the code for the UoSat software port. The remaining code is COTS Linux software. This process would be much the same for any FlightLinux port.

The breadboard architecture includes an asynchronous serial port for debugging. We are utilizing this extensively for debugging . On the spacecraft, the asynchronous port exists, but it is not connected to any additional hardware.

FlightLinux will be implemented in an incremental manner. The initial software build does a "Hello, World" aliveness indication via the asynchronous port and will allow login. Synchronous serial drivers will be integrated to allow communication in the flight configuration. The bulk memory device driver, which uses the 32-megabyte modules of extended memory as a file system, will be added next. The breadboard has a single 32-megabyte module, and there are four such modules in the flight configuration. The CAN bus drivers and the network interface will be added later.

Once we obtain 1) the permission to "export" the software and 2) the SSTL agreement to uplink, we will load a simplified "Hello, World" test kernel in the on-orbit spacecraft for testing. Additional modules can be uplinked later on an incremental basis.

Open-Source Usage

The FlightLinux Project is exploring new issues in the use of "free software" and open-source code, in a mission-critical application. Open-Source code, as an alternative to proprietary software, has advantages and disadvantages. The chief advantage is the availability of the source code, with which a competent programming team can develop and debug applications, even those with tricky timing relationships. The Open-Source code available today for Linux supports international and ad hoc standards. The use of a standards-based architecture has been shown to facilitate functional integration. It is a misconception that "free software" is necessarily available for little or no cost. The "free" part refers to the freedom to modify the source code. The basis of FlightLinux, the BlueCat release of the embedded Linux code, costs $200.

A disadvantage of developing with Open Source may be the perception that freely downloadable source code might not be mature or trustworthy. Countering this argument is the growing experience that the Open-Source offerings are as good as, and sometimes better than the equivalent commercial products. What is needed, however, is a strong configuration control mechanism. For the FlightLinux product, the QSS Team will assume the responsibility of making the "official" version available.

Issues on the development and use of Open-Source software on government-funded and mission-critical applications are still to be explored.

Target Architectures

Various microprocessor architectures have been and are being adapted from commercial products for space-flight use. For all of the primary architectural candidates we identified, Linux is available in COTS form. The primary hardware for flight computers in the near term appear to be derived from the Motorola PowerPC family (RHPPC, RAD6000, RAD750), the SPARC family (EH32), the MIPS family (Mongoose, RH32), the Intel architecture (space flight versions of 80386, 80486, Pentium, Pentium-II, Pentium-III), and the Intel ARM architecture. Versions of FlightLinux for the PowerPC and MIPS family are important goals.

Given the candidate processors identified for missions under development and planned in the short term, we then examined the feasibility of Linux ports for these architectures. In every case, a Linux port was not only feasible, but is probably available as COTS. Each would need to be customized to run on the specific hardware architecture configuration of the target board.

Existing space processors in recent or planned use include the RAD6000, the RH32, and the MIPS-derived Mongoose-V. Generally, Linux requires a Memory Management Unit (MMU) for page-level protection, as well as dynamic memory allocation. However, ports of Linux (uCLinux) exist for the Motorola ColdFire processor series and similar architectures, all without memory management. The ELKS variant of Linux runs on Intel architecture without memory management. The Mongoose architecture does not include memory management hardware. A Mongoose port of Linux is feasible, and this has been examined in conjunction with GSFC, Code 582, Flight Software Branch. The future usage plans of these hardware architectures will determine the direction of our efforts on the FlightLinux software ports. The RAD6000 is reported to be "a direct transfer of the IBM RISC System/6000 single-chip CPU to the Lockheed Martin radiation-hardened process." The RAD/6000 is a PowerPC-like architecture; IBM implemented their later RAD/6000 systems with PowerPC chips. The PowerPC architecture is the result of a joint venture between IBM and Motorola, and incorporates the instruction set of the RAD/6000 line, with the RISC features of the Motorola 88k line.

Emerging space processors include Honeywell's RHPPC, the Lockheed's RAD750, ESA's ERC32, and the Sandia radiation-hard Pentium. All are viable targets for FlightLinux. The RHPPC and the RAD750 are variations of the Motorola PowerPC architecture. GSFC Code 586 already has Linux running on the PowerPC architecture, in a laboratory environment. The Intel (Pentium) version of Linux is the most common and can be found in the Code 586 lab as well. ESA's ERC32 is a variation on the SPARC architecture, and Linux is available for the Sun Sparc architecture. The term COTS should be taken to mean that a commercial version for that processor architecture is available. A specific port for the Flight Computer embedded board would involve coding specific device drivers, reconfiguration, and recompilation of the kernel. Linux is a 32-bit operating system, appropriate for matching the emerging 32-bit class of flight computers.

POSIX

POSIX is an IEEE standard for a Portable Operating System-based Unix. The use of a POSIX-compliant operating system and applications has many benefits for flight software. Among these benefits are: 1) software library reuse between missions and 2) software commonality between ground and flight platforms. For compliant code, the function calls, arguments, and resultant functionality are the same from one operating system to another. Source code does not have to be rewritten to port to another environment. Linux variants are mostly, but not completely, POSIX-compliant. The POSIX standards are now maintained by an arm of the IEEE called the Portable Applications Standards Committee (PASC) with the associated web site http://www.pasc.org/.

POSIX compliance is certified by running a POSIX Test Suite, available from the National Institutes of Standards and Technology (NIST). At the moment, we have no plans for POSIX compliance testing of various variations of Linux, although this is being pursued by GSFC.

The advantages of Linux are numerous, but the requirements for spacecraft flight software are unique and non-forgiving. Traditional spacecraft onboard software has evolved from being monolithic (without a separable operating system), to using a custom operation system developed from scratch, to using a commercial embedded operating system such as VRTX or VxWorks. None of these approaches have proved ideal. In many cases, the problems involved in the spacecraft environment require access to the source code to debug. This becomes an issue with commercial vendors. Cost is also an issue. When source code is needed for a proprietary operating system, if the manufacturer chooses to release it at all, it will be under a very restrictive non-disclosure agreement, and at additional cost. The Linux source code is freely available at the beginning of the effort.

As a variation of Linux, and thus Unix, FlightLinux is Open Source, meaning the source code is readily available and free. FlightLinux currently addresses soft real-time requirements and is being extended to address hard real-time requirements for applications such as attitude control. There is a wide experience base in writing Linux code that is available to tap.

The use of the FlightLinux operating system will simplify several previously difficult areas in spacecraft onboard software. For example, the FlightLinux system imposes a file system on onboard data storage resources. In the best case, Earth-based support personnel and experimenters may network-mount onboard storage resources to their local file systems. The FlightLinux system both provides a path to migrate applications onboard and enforces a commonality between ground-based and space-based resources.

We are pursuing the IEEE POSIX compliance issues of standard embedded Linux, in parallel with an effort in GSFC Code 582, which is collecting a library of POSIX-compliant flight applications software. FlightLinux will also enable the implementation of the Java Virtual Machine, allowing for the up-link of Java applets to the spacecraft.

Linux is not by nature or design a real-time operating system. Spacecraft embedded flight software needs a real-time environment in most cases. However, there are shades of real time, specified by upper limits on interrupt response time and interrupt latency. We can generally collect these into hard real-time and soft real-time categories. Examples of hard real-time requirements would be for attitude control, spacecraft clock maintenance, and telemetry formatting. Examples of soft real-time requirements would include thermal control, data logging, and bulk memory scrubbing.

Unix, and Linux, were not designed as real-time operating systems, but do support multi-tasking. Modifications or extensions to support and enforce process prioritization are necessary to apply Linux to the embedded real-time control world.

In one model, a process may yield the CPU to another pending task. In a preemption scheme, a running process is stopped, and a pending process is started. In another scheme, time slicing, a "round-robin" priority scheme allows equal access to all tasks, or a variation, with a high-priority and a low-priority queue. It is generally agreed that a preemptive scheduling scheme allows for greater concurrency in a real-time system. Beyond the process-switching scheme is the interrupt prioritization. Here, we mean asynchronous interrupts from external sources. Interrupt prioritization is determined and enforced by the hardware configuration. Also, interrupt servicing supersedes software process execution in general.

Problems originate from the fact that the traditional Unix or Linux kernel is a monolithic entity that governs process prioritization. Interrupt drivers and the kernel itself do not participate in the prioritization scheme. The kernel typically has large stretches of non-preemptible code. This is necessarily in the design so that data structures can be modified in an atomic fashion. In a Linux kernel, all interrupt handlers run at a higher priority than the highest-priority task. In the Unix view, the kernel is the top level and most important task. In the real-time control world, this is not necessarily true.

One approach to correcting this is to implement a threaded execution approach for the kernel and the interrupt handlers. The question arises as to how much the Linux kernel can be modified and still be referred to as a Linux kernel. Another approach is to treat the kernel itself as a scheduled task, under a Real Time Task Manager that manages process prioritization and takes over control of interrupts. This approach has been referred to as kernel cohabitation.

At least two real-time schedulers for Linux are available for download. These are a Rate Monotonic Scheduler, which treats tasks with a shorter period as tasks with a higher priority, and an Earliest Deadline First (EDF) scheduler. Other approaches are also possible. It is not clear which approach will prove the best in the spacecraft-operating environment.

Linux is evolving in the direction of full POSIX compliance. The GSFC Flight Software Branch, Code 582, is building a collection of POSIX-compliant application software. The question remains as to how much POSIX-compliance is enough. Complete compliance with the standard for applications and the operating system is probably not required nor warranted.

The Bulk Memory Device Driver

Spacecraft onboard computers do not usually employ rotating magnetic memory for secondary storage. Initially, magnetic tape was used, but now the state of the art is to use large arrays of bulk Dynamic Random Access Memory (DRAM), with various error detection and correction hardware and/or software applied.

A device driver is a low-level software routine that interfaces hardware to the operating system. It abstracts the details of the hardware, in such a way that the operating system can deal with a standardized interface for all devices. In Unix-type operating systems such as Linux, the file system and the I/O devices are treated similarly.

Device drivers are prime candidates for implementation in assembly language because of the need for bit manipulation and speed. They can also be implemented in higher-order languages such as "c" however. Typical device drivers would include those for serial ports, for parallel ports, for the mass storage interface (SCSI, for example), for the LAN interface, etc. Device drivers are both operating system-specific, and specific to the device being interfaced. They are custom code, created to adapt and mediate environments.

The current state of the art for spacecraft secondary storage is bulk memory, essentially large blocks of DRAM. This memory, usually still treated as a sequential access device, is mostly used to hold telemetry during periods when ground contact is precluded. Bulk memory is susceptible to errors on read and write, especially in the space environment, and needs multi-layer protection such as triple-modular redundancy (TMR), horizontal and vertical Cyclic Redundancy Codes (CRC), Error Correcting Codes (ECC), and scrubbing. Scrubbing can be done by hardware or software in the background. The other techniques are usually implemented in hardware. With a Memory Management Unit (MMU), using 1:1 mapping of virtual to physical addresses, the MMU can be used to re-map around failed sections of memory.

Although we usually think of bulk memory as a secondary storage device with sequential access, it may be implemented as random access memory within the computer's address space. This is the approach with UoSat-12.

The Flash File System (FFS) has been developed for Linux to treat collections of flash memory as a disk drive, with an imposed file system. Although we are dealing with DRAM and not flash, we can still gain valuable insight from the FFS implementation. In addition, the implementation of Linux support for the Personal Computer Memory Card International Association (PCMCIA) devices provides another useful model.

The onboard computer on the UoSat-12 spacecraft has 128 megabytes of DRAM bulk memory. It is divided into four banks of 32 megabytes each, mapped through a window at the upper end of the processor's address space. This is the specific device driver that the FlightLinux team will develop and use as a model for future development of similar modules. The current software of the UoSat-12 onboard computer treats this bulk memory as paged random access memory and applies a scrubbing algorithm to counter environmentally induced errors.

The RAM disk is a disk-like block device implemented in RAM. This is the correct model for using the bulk memory of the onboard computer as a file system. Multiple RAM disks may be allocated in Linux. The standard Linux utility "mke2fs," which creates a Linux second extended file system, works with RAM disk and supports redundant arrays of inexpensive disks (RAID) level 0.

The RAID model was developed to use large numbers of commodity disk drives combined into one large, fault-tolerant, storage unit. The approach can be applied to bulk memory as well. RAID can be implemented in software or hardware. For the purposes of this document, we will consider RAID software implementations. Software RAID is a standard Linux feature, available as a patch to the 2.12 Kernels and slated to become an included feature in Kernel 2.4.

This initial version of the driver will use memory mirroring, with memory scrubbing techniques applied. In the simplest case, we will treat three of the four available 32-megabyte memory pages as a mirrored system. The memory scrubbing technique will be derived from the current scheme used by SSTL, as will the paging scheme. The next version of the driver will use all four of the available 32-megabyte memory pages with distributed parity. The performance with respect to write speed is expected to be less than with the Level 0, but the memory resilience with respect to error should be much better.

It is unclear without further testing whether the RAID technique will be sufficient to counter the environmentally induced errors expected in the bulk memory on-orbit. It is generally accepted that RAID is not intended to counter data corruption on the media, but rather to allow data recovery in case of media failure. A defined testing approach will be used with the bulk memory device driver on the breadboard facility. More extensive testing on-orbit with the UoSat-12 spacecraft will be required to validate the approach.

Onboard LAN

Given that the Linux operating system is onboard the spacecraft, support for a LAN becomes relatively easy. Extending the onboard LAN to other spacecraft units in a constellation also becomes feasible, as does having the spacecraft operate as an Internet node.

Interface between spacecraft components is usually provided by point-to-point connections, or a master/slave bus architecture. The use of a LAN onboard is not yet common. This is partially due to the lack of appropriate space-qualified components.

The avionics bus MIL-STD-1553 and its optical derivative, 1773, are commonly used between spacecraft components. This bus, used in thousands of military and commercial aircraft, has a legacy of applications behind it. Also, 1553 is transformer-coupled and dual-redundant, providing a level of failure protection. The raw data rate is 1 megabit-per-second. It is a master/slave architecture.

For point-to-point connections that do not require the complexity of a 1553/1773 connection, a synchronous serial connection such as RS-422/23 with a bit rate of approximately 1 megabit-per-second is typically used.

A LAN-type architecture is typically used in office and enterprise environments (and spacecraft control centers). It provides a connection between peer units, or clients, and servers. The typical LAN uses a coax or twisted pair connection at a transmission rate of 10 megabits per second, a twisted pair connection at 100 megabits per second, or optical at 155 megabits per second, with higher speeds possible.

Usually, a LAN is configured with a repeating hub or a central switch between units. The standard protocol imposed on the physical interface is Transmission Control Protocol /Internet Protocol (TCP/IP), although others are possible (even simultaneously). The TCP/IP protocol has become a favored approach to linking computers around the world. The protocol is supported by Linux and most other operating environments.

The UoSat-12 configuration will allow us to exercise the TCP/IP and CAN bus components of an onboard LAN.

Synopsis

We believe that Linux will be a viable choice for flight computer operating systems. We intend to validate that with extensive breadboard and onorbit testing. We also believe that the Open Source development approach is a viable one for critical software for space missions.

REFERENCES

[1] http://www.gnu.org/licenses/licenses.html